Debugging, Testing and Verification of the Deep Space One Spacecraft Software

SPIN 2000 Workshop

30 August - 1 September 2000

Peter R. Glück

NASA's Jet Propulsion Laboratory

California Institute of Technology

Pasadena, CA

Agenda

- Overview of DS1
- Operational Environment
- Development Environment
- Debugging Strategies
- Testing & Verification
- Case Study 1
- Case Study 2
- Conclusion

Overview of DS1

- First Deep Space Mission of the New Millennium Program
  - Demonstrate 12 new technologies including
    - Solar-Electric (Ion) Propulsion
    - Miniature Integrated Camera and Spectrometer
    - Small Deep Space Transponder
  - Three technologies are software-based:
    - Autonomous Navigation
    - Remote Agent Autonomous Planning and Control
    - Beacon-Mode Operations
  - Rendezvous with an asteroid and comet

Software Operational Environment

- IBM RAD6000 Flight Processor
  - Radiation-hard version of the RS6000
  - Similar to PowerPC architecture(RISC)
- VxWorks Real-Time Operating System (RTOS)
  - Version 5.1.1
  - Priority based-scheduling of 53 independent tasks (processes or threads)
- VME (VersaModule Eurocard) avionics backplane
- "Hard" Real-Time Operation Required
  - Most tasks operate at or above 1 Hz
  - Attitude Control runs at 8Hz
  - Failure to maintain real-time control may cause loss of mission

Software Development Environment

- Code development and implementation on Unix Workstations
  - GreenHills and GNU compilers used for initial development and debugging
  - Final build for the RAD6000 on IBM AIX workstations
    - Required due to Mars Pathfinder heritage and short development schedule
- Unit-level testing and debugging on PowerPC (PPC603) real-time processors (VME) (Unit Level Testbed, a.k.a. Babybed)
    - Similar architecture to the RAD6000

Software Development Environment (cont'd)

- System-level testing and debugging on commercial and engineering-model avionics (Flight System Testbed, a.k.a. Papabed)
- Closed-loop hardware and environmental simulation via VME shared memory
- Limited debugging tools (Multi, VxGDB), especially in system-level testing

Debugging Capabilities (cont'd)

- Visual tools:
  - VxGDB (GNU compiler)
  - Multi (GreenHills compiler)
- Software Facilities
  - Event Reports
    - Inherited capability from Mars Pathfinder
    - Prints to screen in test mode, downlinks through telemetry during flight
    - Still a primary means of determining software performance and behavior
  - Special debugging code
    - Logs, printf(), global variables, test routines
    - Compiled-out or even deleted prior to final delivery

Debugging Capabilities

- VxWorks Facilities (pre-Tornado)
  - Memory access (dump, modify)
  - Scripts
  - Global variable access

Testing & Verification

- Testing and verification occur at five levels:
  - Unit-level testing
    - Conducted on a Unit Level Testbed
    - Some core software functions provided in a Unit Test Suite
    - Occasionally performed directly on workstations
  - Module Acceptance Testing
    - Conducted on a Unit Level Testbed
    - Review and regression testing of Unit Test
    - Code inspection

Testing & Verification (cont'd)

- Testing and verification occur at five levels (cont'd):
  - Flight Software Integration Testing
    - Usually the first place that modules (tasks) meet
    - Problems at this stage due primarily to:
      - Mismatch with simulation interfaces
      - Mismatch between module interfaces
      - Mismatch with hardware interfaces
      - Faulty or incorrectly understood hardware
    - Conducted on the Flight System Testbed
  - Avionics System Integration Testing
    - Final integration with flight avionics
    - Occasional problems due to faulty or incorrectly understood hardware

Testing & Verification (cont'd)

- Testing and verification occur at five levels (cont'd):
    - Full-up Spacecraft Testing
        - Tests the integrated system.
        - Main problems here are from phasing errors, unanticipated operational scenarios, and incorrectly understood mission requirements

Case Study 1: Dual-Task Deadlock

- Background
  - Two telemetry-related tasks are the Event Reporting (EVR) task and the Engineering, Housekeeping and Accountability (EHA) task. The EVR task reports on interesting or serious events in the system using four severity levels: INFO, WARNING, FAULT, and FATAL. The EHA task maintains a database of engineering measurements and periodically samples and downlinks the measurements as specified by the operators. These modules were both inherited from the Mars Pathfinder mission.
  - On DS1 we discovered that we were often generating many more EVRs than the system could accommodate for downlink. Once the fixed-size queues filled up EVRs would be lost on the input side of the queue. The quantity of EVRs lost was often important in analyzing problems. In order to track the lost EVRs, new EHA database entries (measurements) were added, causing the EVR publication routine to access the EHA database insertion routines.
  - The EHA database insertion routines, on the other hand, would access the EVR publication routines to indicate an interesting or problematic event had occurred in accessing the database. This was the inherited design.
  - You can probably see where this is going...

Case Study 1: Dual-Task Deadlock

- Symptoms
  - The problem was manifested by a mysterious hanging of the system. Subsequent retrieval of EVRs showed two EVRs occurring over and over again:
    - Error writing to queue: ID = x
    - Illegal producer code y for measurement x
- Analysis
  - There was no apparent reason for the producer code for measurement X to be illegal. That was ultimately attributed (but unproven) to be related to some sort of memory corruption, which was a somewhat common source of errors at the time (e.g., from an improperly initialized pointer or variable, or accessing beyond an array boundary).
  - In this case it turns out that it was the producer code of the measurement indicating the number of FATAL EVRs that was corrupt. A call to update this EHA measurement would thus generate another FATAL EVR message, which would then cause another call to EHA, ad infinitum.
  - Further complicating this problem was the fact that the original FATAL EVR appeared to have been generated from the interrupt-level task context, which has the unique properties of being the highest priority task in the system and also being unsuspendable.

Case Study 1: Dual-Task Deadlock

- Analysis (Cont'd)
  - What was apparent was that the introduction of the new lost-EVR measurements had the unintended effect of deadlocking the system under the following conditions:
    1. The FATAL EVR message queue must be overflowing
    2. An EHA error must occur to generate an FATAL EVR message from within the EHA database insertion routine.
    3. A FATAL EVR message must be occurring from interrupt context (otherwise the affected task would simply be suspended after the first EVR generation).
- Solution
  - One could argue that the root problem was in the two-way dependence of the EHA and EVR tasks. However, that is not really the case. The problem was in the way the dependence was set up. The change had been implemented (by yours truly) so that whenever the queuing of a new EVR failed a call was made to update the EHA database. It turns out that it is not really necessary to update the EHA database every time a new EVR fails to be inserted, because the EHA database is sampled at a fairly low rate (typically between 5 and 20 seconds). It is therefore sufficient to simply update a counter and periodically send the new counts to the EHA database. So, the solution was to simply move the EHA database insertion out of the EVR reporting routine and into a separate task context that periodically updated the measurements.

Case Study 2: Downlink Handshake Disruption

- Background
  - Commands may be uplinked to the spacecraft for immediate execution or future execution. Commands slated for future execution are normally sent in groups of commands called command sequences. When a new sequence is uplinked, the software will immediately validate all commands in the sequence prior to storing the sequence for future execution. This ensures that no corrupt commands are stored in on-board sequences.
  - One such sequence is referred to as the "Telecom Backbone" sequence. This sequence contains a set of telecommunications commands and is intended to be run over and over every three days or so. The Telecom Backbone sequence contains many commands for the spacecraft to construct and downlink time-correlation telemetry packets, which are used to maintain the correlation between the spacecraft time reference clock (which drifts over time) and ground truth time.
  - When data is downlinked to the ground, two separate processes are involved. The first, called simply the Downlink task, provides routines for the creation and storage of telemetry packets to be downlinked and then manages the transmission of those packets. This task runs at a fairly low priority in the system. The second, called the Downlink FIFO task, runs at high priority and exists to service the hardware that modulates the data onto the radio frequency carrier signal. Whenever the Downlink FIFO task puts packets into the hardware, it also sends a request back to the Downlink task to clean up the packet storage area. This is referred to as the "purge-by-age" message.

Case Study 2: Downlink Handshake Disruption

- Symptoms
  - 120 EVRs were observed indicating that the Downlink task Inter-Process Communication (IPC) message queue had overflowed. All were issued within the same second.
  - No other obvious symptoms were observed.
  - THIS PROBLEM OCCURRED IN FLIGHT!
- Analysis
  - The Downlink IPC queue was sized to handle 25 messages. No other EVRs were issued during the interval. The first 119 of these EVRs were issued by the Uplink task, while the 120th EVR was issued by the Downlink FIFO task.
  - The cause of these EVRs was a software bug in the Command dispatch task that resulted in execution of the make_time_packet() function during the sequence validation (which occurs in the Uplink task context). The sequence contained 144 MAKE_TIME_PACKET commands; since these commands were then executed in rapid succession they were all queued up on the Downlink IPC queue, resulting in an overflow of the queue and 144-25=119 EVRs associated with the (discarded) attempts to execute the MAKE_TIME_PACKET command. The 120th EVR came from the Downlink FIFO task. This is believed to have been a purge-by-age message that is routinely sent by the Downlink FIFO task whenever it processes a packet.

Case Study 2: Downlink Handshake Disruption

- Analysis
  - It was initially believed that loss of the purge-by-age message was harmless because a new message is generated for each downlink frame. However, the message handling logic includes a handshake that, if skipped (due to a dropped message), will result in the purge-by-age function being effectively disabled because an internal flag indicating that the message has been handled never gets reset. Thus until this flag is reset the purge by age function will not do anything and the packet buffers will become clogged with old, sent telemetry packets.
  - The conditions necessary to encounter this problem are not difficult to produce. All that is required is to request Downlink to process more than 25 packets (or other messages) at once and you will probably (depending on the downlink data rate) have an opportunity to drop the purge-by-age message, especially since Downlink operates at a fairly low task priority.
- Solution
  - Implement more robust handshaking that is not sensitive to the loss of one or more purge-by-age messages.

Future Directions

- Spacecraft avionics are moving to more commercialized hardware, such as the RAD750
  - Enables development on inexpensive equivalent equipment, e.g., commercial PPC750 boards
    - In contrast, Mars Pathfinder developed code on M68K boards but flew on a RAD6000
  - Provides for better vendor support for compilers and operating systems due to the larger user community
    - DS1 options were limited due to lack of vendor interest in supporting the RAD6000
  - Permits use of available COTS tools (e.g., Tornado tools for VxWorks)

Future Directions (cont'd)

- A caution about reliance on integrated tools:
  - Once the spacecraft launches, integration with tools becomes problematic, particularly for deep space missions
  - While TCP/IP protocols for spacecraft have been proposed, the transmission latency introduced by the one-way light-time between the spacecraft and the user make interactive tools impractical
  - Therefore, traditional methods of debugging spacecraft problems (e.g., event reports) will continue to be necessary and important in operating deep space missions
    - Use of these methods must begin early in the integration and test phase to become familiar with them
  - New tools will augment, rather than replace, traditional tools

Conclusion

- Space missions place unique requirements on software development:
  - Radiation-hard processors, hard-real-time processing, Extremely robust fault responses, remote operation
- These requirements often demand that less sophisticated tools be used
  - Simply not available for the target processors
  - In-flight debugging capability is required
- They also often restrict developers from making full use of language features
  - For example, use of pointers and dynamic memory allocation in 'C'-based languages is a common source of latent defects

Conclusion (cont'd)

- Yet, many of the same problems of other multithreaded systems exist
  - Deadlock issues, dropped handshakes, synchronization, query-response
- Tools that can assist in detection of these issues without undue effort have the potential for reducing spacecraft flight software development cycle time and costs
  - Could allow problem-detection efforts to focus on integration with hardware rather than software-internal defects
  - However, traditional methods (e.g., event reports) will still be necessary due to the need for remote debugging post-launch